Implementation of Insertion Sort into Merge Sort Using Java

Homework #3

By:

Pratyusha Thundena

CS 303L – L1A

September 19, 2018

### Problem Specification

The objectives included the following: applying the merge sort algorithm and analyzing the performance of merge sort with respect to array size, comparing the recorded time of insertion sort with the recorded time of merge sort (for the same input files), creating a revised merge sort program that implements insertion sort when the data input is small, and evaluating the differences between the two algorithms based on the given output.

### Program Design

Two programs were created to meet the objectives of this assignment. The mergeSort.java program ran the mergeSort algorithm without implementing the insertionSort algorithm. Meanwhile, the mergeSortNew.java program included a conditional statement that utilized the insertion sort algorithm when the input array was small. Both programs required the following: text file with 100 random numbers (input\_100.txt), text file with 1,000 random numbers (input\_1000.txt), text file with 5,000 random numbers (input\_5000.txt), text file with 10,000 random numbers(input\_10000.txt), text file with 50,000 random numbers(input\_50000.txt), text file with 100,000 random numbers (input\_100000.txt), text file with 500,000 random numbers (input\_500000.txt), Scanner class, Arrays class, File class, and FileNotFoundException class.

The following steps were required to develop the mergeSortNew.java program:

1. scan the following text files: input\_100.txt, input\_1000.txt, input\_5000.txt, input\_10000.txt, input\_50000.txt, input\_100000.txt, and input\_500000.txt
2. save the text files in a known variable
3. read each text file
4. proceed through the conditional clause containing insertionLimit
5. perform either merge sort or insertion sort algorithm for the array based on the conditional statement
6. sort the elements of each text file
7. record the time taken to sort the elements of all the input files
8. display the recorded time for each sorted text file

\*The same steps were required to create the mergeSort.java program with the exception of step d.

The following methods and constructors were defined for both programs:

1. merge\_sorting ()

Basic method that performs the conditional clause (i.e. contains insertion limit) to determine whether or not insertion sort should be performed on the input array.

b) merge ()

Basic method that contains the merge sort algorithm.

c) main ()

Driver method used to display information to the user.

d) insertionSort ()

Basic method that contains the insertion sort algorithm.

d) Scanner ()

Constructor used to read necessary values with the provided driver program.

\*The println method of the System.out object was used to display the inputs and results for the driver program.

### Testing Plan

The first program (mergeSort.java) tested the following input sizes: 100, 1000, 5000, 10000, 50000, 100000, and 500000. Then the time was recorded for each array. The second program (mergeSortNew.java) tested the same input sizes as the first program. Unlike the first program,

this program was used for insertion sort when the data set was small (i.e. 100 elements). Then the time was recorded for the sorted array that went through insertion sort. The time was also recorded for the multiple sorted arrays that progressed through merge sort.

### Results

**Figure A: mergeSort.java program**

|  |  |
| --- | --- |
| **Text File** | **Recorded Time to Sort File (in milliseconds):** |
| input\_100.txt | 0.0 |
| input\_1000.txt | 15.0 |
| input\_5000.txt | 71.0 |
| input\_10000.txt | 10.0 |
| input\_50000.txt | 317.0 |
| input\_100000.txt | 1709.0 |
| input\_500000.txt | 54380.0 |

**Figure B: mergeSortNew.java program**

|  |  |  |
| --- | --- | --- |
| **Sort Method** | **Text File** | **Recorded Time to Sort File (in milliseconds):** |
| Insertion | input\_100.txt | 6.0 |
| Merge | input\_1000.txt | 20.0 |
| Merge | input\_5000.txt | 132.0 |
| Merge | input\_10000.txt | 8.0 |
| Merge | input\_50000.txt | 341.0 |
| Merge | input\_100000.txt | 1598.0 |
| Merge | input\_500000.txt | 52959.0 |

### Analysis and Conclusions

In the first program (Figure A), the pattern was that the larger array took longer to sort than the smaller array. The second program (Figure B) demonstrated a similar pattern. Moreover, insertion method was used when there were 100 elements. If the insertion method was utilized for 1000 elements, then it was plausible that the time would be too fast for an accurate comparison between merge sort and insertion sort. For this reason, the cutoff was input\_1000.txt. Another important point (to note) was that the size of an array was not always as important as how far the array was from its sorted state. For mostly ordered elements, insertion sort could run more quickly than merge sort. If the array was large and unordered, then merge sort was the preferred method for sorting the array. In this particular assignment, the mergeSort algorithm with the average time complexity of O(nlog(n)) was remarkably faster than the insertionSort algorithm with the average time complexity of O(n2).

### References

The input files provided in the homework assignment and the sample report provided by Dr. Purushotham Bangalore was used to do the lab report.